WarpX

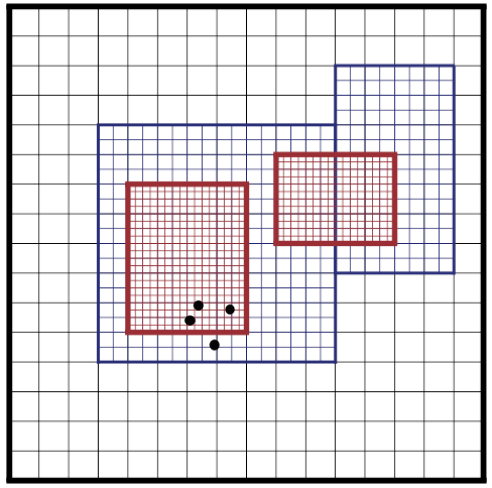
Structure of the code, and how to navigate through the source

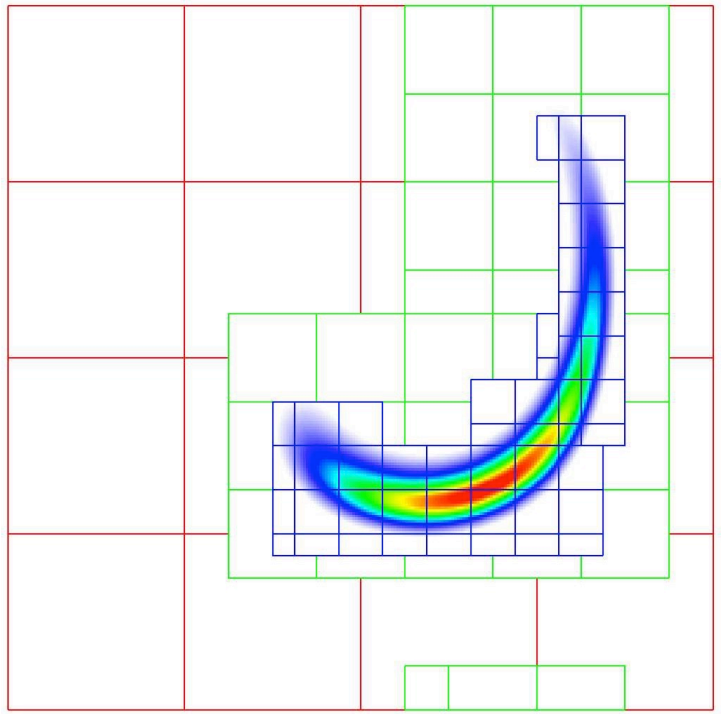
Maxence Thévenet
LBNL
03/05/2020

Part I: code organization
- Main classes
- Main steps in a simulation
- Zoom on specific capabilities
- How to be a good citizen-contributor

AMReX basics: block-structured mesh refinement

Box	Lower and upper indices
FArrayBox	Array defined on a Box
FABArray/MultiFAB	Collection of FArrayBox
ParticleContainer	Collection of particles per box, with an iterator





Iterator over Box/tile

```
for (WarpXParIter pti(*this, lev); pti.isValid(); ++pti){
    auto& attribs = pti.GetAttribs();
    auto& uxp = attribs[PIdx::uxp];
    const FArrayBox& exfab = Ex[pti];
```

Particle attributes (SoA)

Array associated with the box/tile

Main WarpX classes (everything in Source/, .H file with same name as class)

"Multi" = "collection of"

BIG AMReX dependency:
- Domain decomposition (Box, BoxArray, DistributionMapping, Geometry)
- Structures for fields (MultiFab) and particles (ParticleContainer)
- Communications (FillBoundary, Redistribute)
- Portability (ParallelFor)

WarpX : AmrCore	./	All classes, Fields (= MultiFab), Full diags (plotfiles), time steps, some BC
WarpXParticleContainer : ParticleContainer	Particles/	1 species (physical or not), per box, based on Particle, FG, PP, CD
MultiParticleContainer	Particles/	Vector of all species, and loops over species
FiniteDifferenceSolver	FieldSolver/FiniteDifferenceSolver/	Evolve E and B (CKC, Yee, orders?)
SpectralSolver	FieldSolver/SpectralSolver/	Evolve E and B (PSATD, etc.)
ILaserProfile, GaussianLaserProfile etc.	Laser/	Laser profile (Gaussian, from file, parsed, etc.)
LaserParticleContainer	Laser/	Antenna
PlasmaInjector	Initialization/	Plasma profile
WarpXParser	Parser/	General parser (used in many places)
ReducedDiagnostics	Diagnostics/ReducedDiags/	Reduced diags, well-separated module
BackTransformedDiagnostics	Diagnostics/	Big machinery, for runs in a boosted frame
GuardCellManager	Parallelization/	Grid communications. uses on MultiFab::FillBoundary
PML	BoundaryConditions/	Well-separated module, BC
Filter	Filter/	NCI filter (on E and B), bilinear filter (on J)

particles
Field solver
Initial conditions
Diagnostics
Communications
Boundary condition
Numerics

WarpX: Fields

WarpX has a LOT of fields, all of them are members of class WarpX

- E
- B
- J
- rho

- Fine patch
- Coarse patch
- Auxiliary
- Coarse-aux

- Physical space
- Fourier space

`Efield_fp`

All field arrays are directly members of the class WarpX

```
class WarpX
    amrex::Vector<std::array< std::unique_ptr<amrex::MultiFab>, 3 > > Efield_fp;
```

MR levels

Component

We use the `amrex::MultiFab` member functions for most of our operations:

- Communication
- Interpolation
- Basic operations

Most functions have a general and a per-level version

```
void
WarpX::EvolveE (amrex::Real a_dt)
{
    for (int lev = 0; lev <= finest_level; ++lev) {
        EvolveE(lev, a_dt);
    }
}
```

```
void
WarpX::EvolveE (int lev, amrex::Real a_dt)
{
    m_fdtd_solver_fp[lev]->EvolveE( Efield_fp[lev], etc. );
}
```

why not `amrex::Vector<std::unique_ptr<amrex::MultiFab> > Efield_fp;` with 3-component MultiFabs?

WarpX: Particles 1/2 : Particles and ParticleContainers

`amrex::Particle`
`amrex::ParticleContainer`

p.pos(0) = x position of the particle

- `WarpXParticleContainer` : `amrex::ParticleContainer`
- `LaserParticleContainer` : `WarpXParticleContainer`
- `PhysicalParticleContainer` : `WarpXParticleContainer`
- `PhotonParticleContainer` : `PhysicalParticleContainer`
- `RigidInjectedParticleContainer` : `PhysicalParticleContainer`

ParticleContainer = species
Particles are stored per-box (or per-tile on CPU)

```
Array-of-Struct: x1 y1 z1 x2 y2 z2 … xn yn zn   : position, id, CPU
Struct-of-Array: ux1 ux2 … uxn … uz1 uz2 … uzn : momentum, fields, etc.
```

STANDARD WAY

```
for (WarpXParIter pti(*this, lev); pti.isValid(); ++pti){
    const auto GetPosition = GetParticlePosition(pti, offset);
    auto& attribs = pti.GetAttribs();
    auto& uxp = attribs[PIdx::ux];
    const long np = pti.numParticles();
    amrex::ParallelFor(np, …
        amrex::ParticleReal xp, yp, zp;
        GetPosition(ip, xp, yp, zp);
        // now I can play with xp, uxp[ip], etc.
    );
```

NEW WAY

```
using PType = typename
WarpXParticleContainer::SuperParticleType;
Ptype p;
p.pos(0);
p.rdata(PIdx::w);
```

WarpX: Particles 2/2: MultiParticleContainer

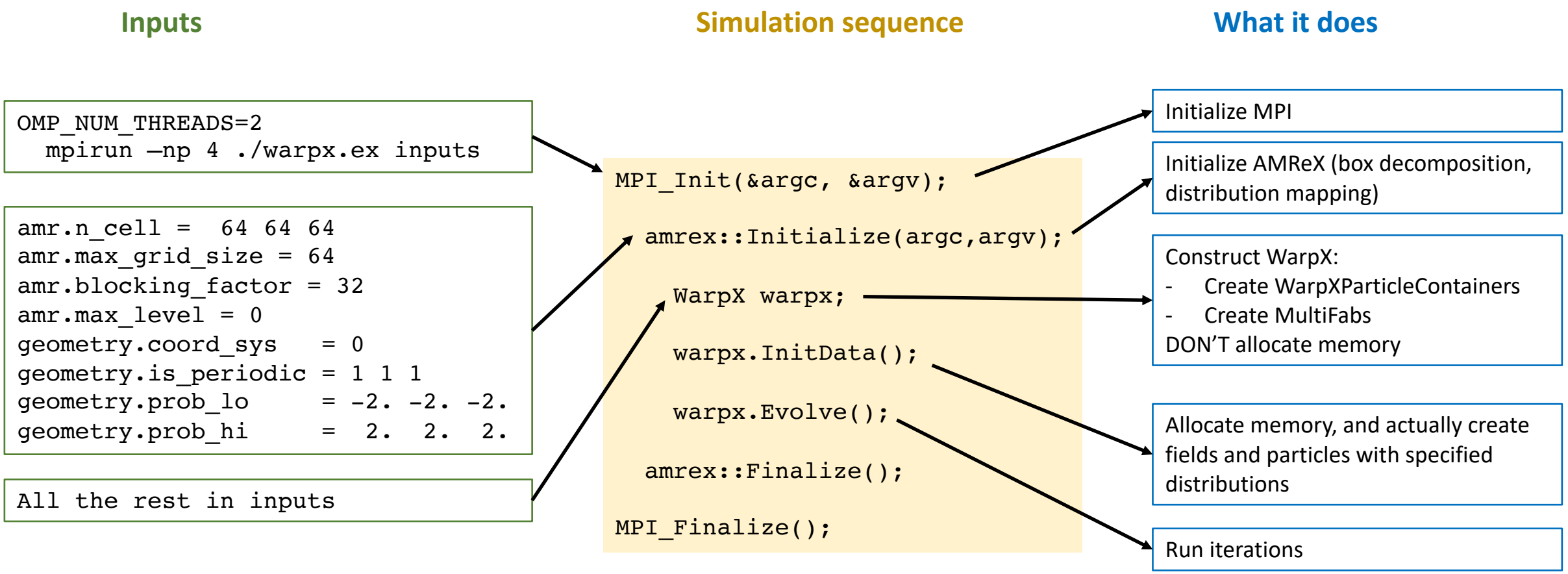
1 ParticleContainer = 1 species.
Class that collects all ParticleContainers (= all species): MultiParticleContainer

```
class MultiParticleContainer
    amrex::Vector<std::unique_ptr<WarpXParticleContainer> > allcontainers;
    multi-species handling (QED, etc.)

    void
    MultiParticleContainer::FieldGather ()
    {
        for (auto& pc : allcontainers) {
            pc->FieldGather(lev, Ex, Ey, Ez, Bx, By, Bz);
        }
    }
```

Class WarpX has a SINGLE MultiParticleContainer member variable: mypc

```
class WarpX
    std::unique_ptr<MultiParticleContainer> mypc;
```

What happens when I run a simulation? Overview



What happens when I run a simulation? Initialization sequence

```
WarpX warpx;

warpx.InitData();
```

WarpX::WarpX

- WarpX::ReadParameters
- MultiParticleContainer::MultiParticleContainer
  - MultiParticleContainer::ReadParameters
  - Loop over all species
    - ParticleContainer::ParticleContainer
      - ParticleContainer::ParticleContainer
      - ParmParse::query()

Warpx.do_filter

particles.species_names = electrons

electrons.num_particles_per_cell = 2

Null pointers, no particles. Advice?

WarpX::InitData

- WarpX::InitFromScratch
  - AmrCore::InitFromScratch
    - WarpX::MakeNewLevelFromScratch
      - WarpX::AllocLevelData
        - WarpX::AllocLevelMF
      - WarpX::InitLevelData
        - MultiFab::setVal
  - MultiParticleContainer::AllocData
  - MultiParticleContainer::InitData
    - PhysicalParticleContainer::AddParticles(0)
      - PhysicalParticleContainer::AddPlasma
    - ParticleContainer::Redistribute
  - WarpX::ComputeSpaceChargeField
  - WarpX::InitPML
    - PML::PML
- WarpX::InitFilter
  - Filter::Filter
- WarpX::InitDiags (BTD)

Allocate all field MultiFabs!

Inject particles with requested plasma profile

What happens when I run a simulation? Time iteration sequence

`warpx.Evolve();`

WarpX::EvolveEM (Evolve): loop over iterations

- MultiParticleContainer::PushP If needed, ½ push back for particle momenta
- UpdateAuxilaryData();
- **WarpX::OneStep_nosub**
- BackTransformedDiagnostics::writeLabFrameData
- MoveWindow
- MultiParticleContainer::Redistribute
- MultiParticleContainer::SortParticlesByBin
- ReducedDiags::ComputeDiags
- WarpX:: WritePlotFile

WarpX::OneStep_nosub **PIC loop**

```
mypc->doFieldIonization();
mypc->doCoulombCollisions();
PushParticlesandDepose();
SyncCurrent();
EvolveB(0.5*dt[0]);
FillBoundaryB();
EvolveE(dt[0]);
FillBoundaryE();
EvolveB(0.5*dt[0]);
if (do_pml)
    DampPML();
    FillBoundaryE();
    FillBoundaryB();
```

WarpX::PushParticlesandDepose: loop over ParticleContainers and call **WarpXParticleContainer::Evolve**

```
for (WarpXParIter pti(*this, lev); pti.isValid(); ++pti)
    applyNCIFilter
    FieldGather
    PushPX
    DepositCurrent
```

Topics treated below

- Portability
- PIC elementary routines
- Communications
- Diagnostics
- MR
- CI
- Profiling
- PR

Portability: AMReX::ParallelFor

Loop over particles: CPU

```
for(int i=0; i<N; i++){
    xp[i] += 1.;
}
```

Loop over particles: GPU

```
CUDA (NVIDIA)
kernel(int* xp) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i<N) xp[i] += 1.;
}
kernel<<<N, 256>>>(xp);
```

We don't want to write everything twice. AMReX provides an abstraction layer (portability layer), its main function is ParallelFor
We write the code only once, and it is compiled as a for loop or kernel launch, depending on USE_GPU

```
amrex::ParallelFor(
    N,
    [=] (int i) {
        xp[i] += 1.;
    }
);
```

We cannot use anything inside a ParallelFor!!

- std::vector is NOT going to work well on GPU
- If you use a member variable (of class C) in a parallel for, the whole class will be copied to the GPU. WE DON'T DO THAT (except for small classes that we know are GPU-friendly)
  → Make a local copy of just the variable, and use this copy

AMReX provides different flavors of ParallelFor, for loops on particles, fields, etc.

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

```
amrex::ParallelFor(tex, tey, tez,
    [=] AMREX_GPU_DEVICE (int i, int j, int k){
        Ex(i, j, k) += 1.;},
    [=] AMREX_GPU_DEVICE (int i, int j, int k){
        Ey(i, j, k) += 1.;},
    [=] AMREX_GPU_DEVICE (int i, int j, int k){
        Ez(i, j, k) += 1.;}
);
```

PIC elementary functions

`Particles/Gather/FieldGather.H`

`void doGatherShapeN`

free-standing function, called in `PhysicalParticleContainer::Evolve`

`Particles/Pusher/UpdateMomentumBoris.H`

`void PhysicalParticleContainer::PushPX`

Member of PhysicalParticleContainer, called in `PhysicalParticleContainer::Evolve`

`Particles/Deposition/CurrentDeposition.H`

`void doDepositionShapeN`

free-standing function, call in `PhysicalParticleContainer::Evolve`

`FieldSolver/FiniteDifferenceSolver/EvolveE.cpp`

`void FiniteDifferenceSolver::EvolveE`

member of `FiniteDifferenceSolver,` called in `WarpX::OneStep_nosub`

Communications

A PIC code should only require local communications between neighbor boxes.
Comm patterns can be tricky with MR, though → AMReX handles ALL of it.

Field halo exchanges: FillBoundary. Called many times per iteration.
In WarpX, we try to exchange as little data as possible. GuardCellManger keeps track of the number of cells we need to exchange at each part of the PIC code. That could be improved (in particular for PMLs).

Particle halo exchanges: Redistribute. Called once per iteration.

```
WarpX::OneStep_nosub                                    PIC loop
    mypc->doFieldIonization();
    mypc->doCoulombCollisions();
    PushParticlesandDepose();
    SyncCurrent();
    EvolveB(0.5*dt[0]);
    FillBoundaryB();
    EvolveE(dt[0]);
    FillBoundaryE();
    EvolveB(0.5*dt[0]);
    if (do_pml)
        DampPML();
        FillBoundaryE();
        FillBoundaryB();
```

Diagnostics

Full diagnostics (plotfiles): Part of class WarpX ☹

Mostly in Diagnostics/

Reformatting in progress

```
WarpX::EvolveEM
→ WarpX::WritePlotFile
    → WarpX::prepareFields
        → WarpX::AverageAndPackFields
        → WarpX::coarsenCellCenteredFields
    → Amrex::WriteMultiLevelPlotfile (or OpenPMD!)
    → WarpX::WriteRawField
    → MultiParticleContainer::WritePlotFile
    → WarpX::WriteJobInfo
    → WarpX::WriteWarpXHeader
```

Slice diags: ~parallel implementation

Reduced diagnostics (~1 number per diag iteration) ☺☺

ALL in Diagnostics/ReducedDiags/

```
class WarpX
    MultiReducedDiags* reduced_diags;
class MultiReducedDiags
    std::vector<std::unique_ptr<ReducedDiags>> m_multi_rd;
class ReducedDiags
```

base class for these reduced diagnostics

```
WarpX::EvolveEM
    → reduced_diags->ComputeDiags(step);
    → reduced_diags->WriteToFile(step);
```

Then, the instance of `ReducedDiags` gets a pointer to WarpX to access all the data (unfortunately in a non-const fashion)

Back-transformed diagnostics (boosted-frame) ☺

Mostly in Diagnostics/

```
Class BackTransformeDiagnostic
   std::vector<std::unique_ptr<LabFrameDiag> > m_LabFrameDiags_;
Class LabFrameDiag
    std::unique_ptr<amrex::MultiFab> m_data_buffer_;
WarpX::EvolveEM
→ GetCellCenteredData
    → AverageAndPackVectorField
    → AverageAndPackScalarField
→ LabFrameDiags::writeLabFrameData
    → LabFrameDiags::LorentzTransformZ
    → … some more stuff with buffers
```

1 LabFrameDiags = 1 snapshot in lab frame (t=t1, z = [zmin, zmax])

Corresponds to different (t', z'): at each iteration:

- take the (z', t') slice that belongs to a given LabFrameDiags, and cell-center it
- Lorentz-transform it (to the lab)
- stack it in a buffer

Diags should be reorganized soon(ish)!

Mesh Refinement

PIC loop without MR

```
PushParticlesandDepose()
     FieldGather()
     PushPX()
     DepositCurrent()
SumBoundaryJ()
EvolveB(0.5*dt[0])
FillBoundaryB()
EvolveE(dt[0])
FillBoundaryE()
EvolveB(0.5*dt[0])
FillBoundaryB()
```

PIC loop with MR

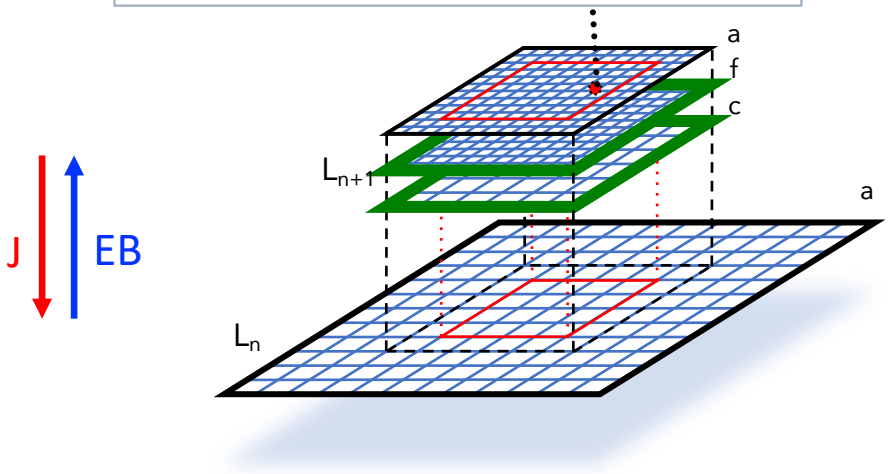
```
UpdateAuxilaryData()
PushParticlesandDepose()
     FieldGather()
     PushPX()
     DepositCurrent()
SyncCurrent()
EvolveB(0.5*dt[0])
FillBoundaryB()
EvolveE(dt[0])
FillBoundaryE()
EvolveB(0.5*dt[0])
FillBoundaryB()
```

```
UpdateAuxilaryData()
     UpdateAuxilaryDataStagToNodal
          MultiFab::setVal
          MultiFab::ParallelCopy
          MultiFab::Substract
```

```
SyncCurrent()
     interpCurrentFineToCoarse()
     AddCurrentFromFineLevelandSumBoundary()
          ApplyFilterandSumBoundaryJ()
          MultiFab::ParallelAdd()
```

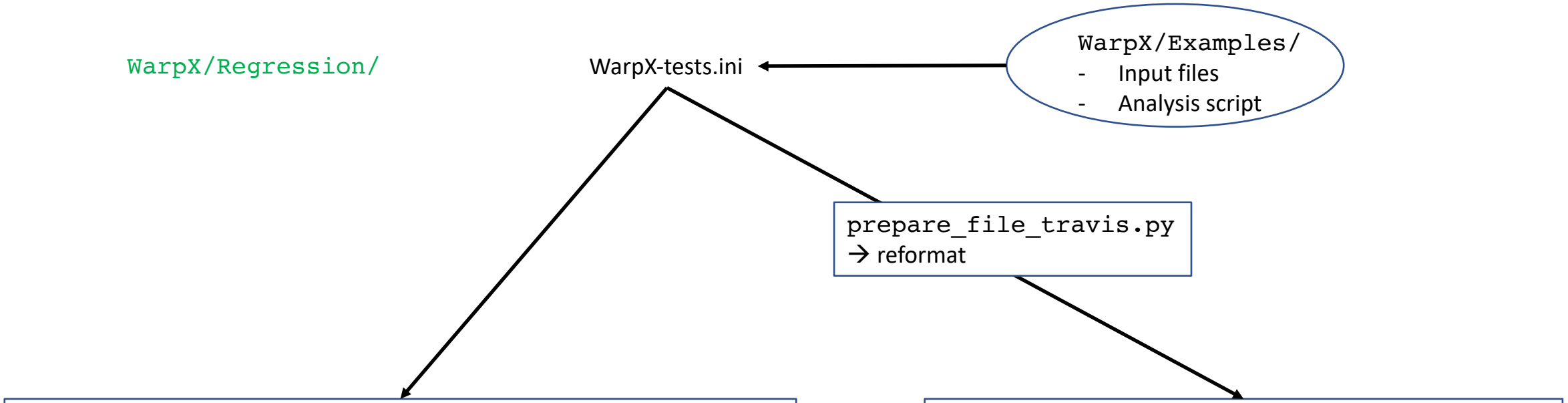
Substitution:

$$F^{n+1}(a) = I[F^{n}(a)-F^{n+1}(c)]+F^{n+1}(f)$$



absorbing layer (PML)
a = auxilliary
f = fine
c = coarse

Continuous Integration



Nightly builds on CRD clusters Battra (CPU) and Garuda (GPU)

- Every night
- See https://github.com/ECP-WarpX/regression_testing
- Compare with ref to machine precision
- Published at https://ccse.lbl.gov/pub/RegressionTesting/WarpX/

→ Catch everything (and more)

TravisCI tests on every commit on GitHub

- Every time you push on a branch with open PR
- GitHub tells you when they fail
- Jobs are submitted by batch (see .travis.yml)
- Only tests compilation, run and analysis!!

→ Only catch what we ask for

Also performance tests but, heh

Profiling in WarpX

```
----------------------------------------------------------------------------------------
Name                                        NCalls  Excl. Min  Excl. Avg  Excl. Max   Max %
----------------------------------------------------------------------------------------
PPC::FieldGather                               800      29.06      29.23      29.32  68.63%
PPC::Evolve()                                  100      8.175      8.264      8.329  19.50%
FabArray::ParallelCopy()                       600       1.11      1.239      1.434   3.36%
FillBoundary_finish()                         1200     0.6782     0.8679       1.03   2.41%
Redistribute_partition                         102     0.8006     0.8529     0.8762   2.05%
FillBoundary_nowait()                         1200     0.4906     0.6733     0.8346   1.95%
```

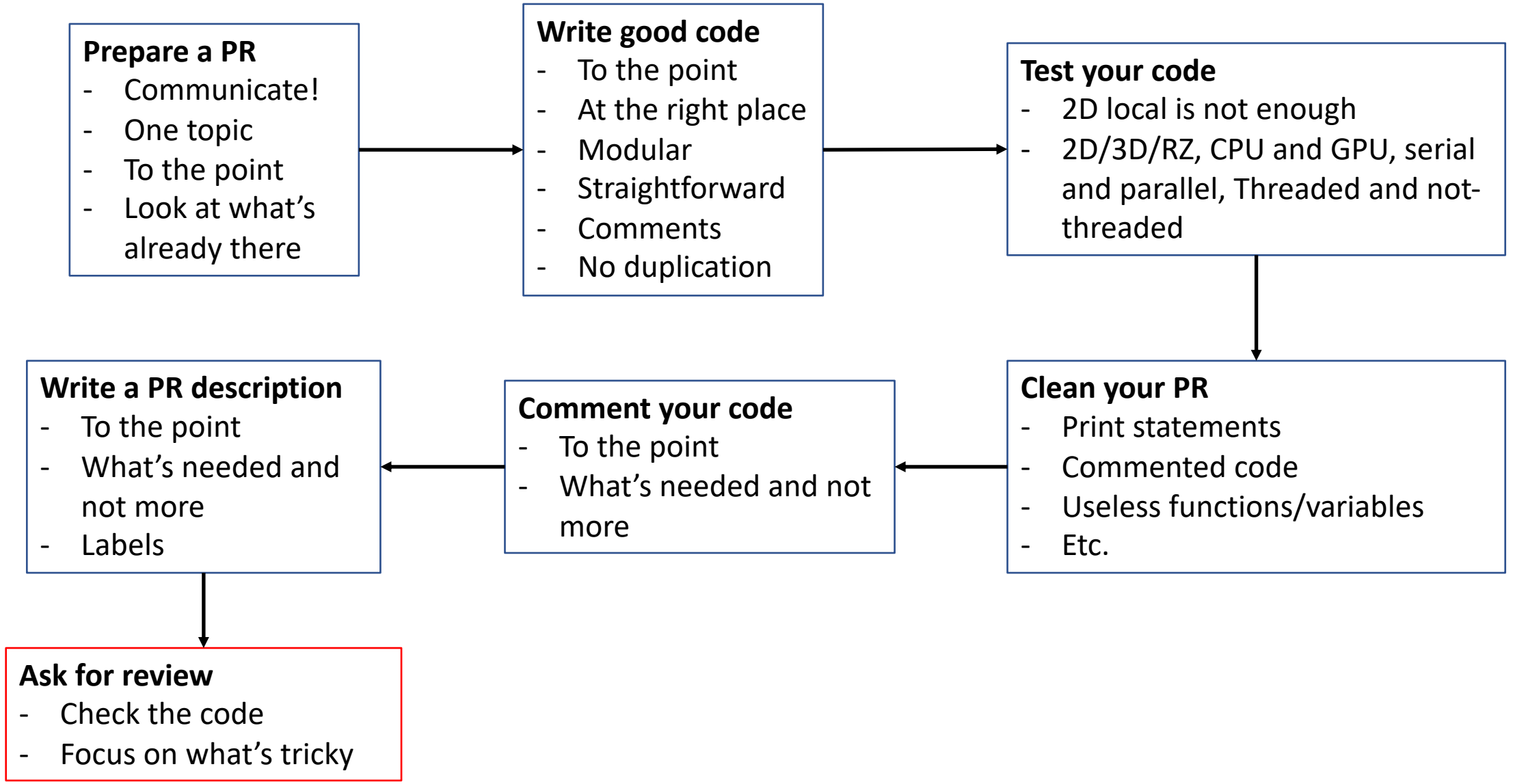
AMReX Tiny Profiler
BL_PROFILE

We have a wrapper in WarpX:
WARPX_PROFILE

```
void PhysicalParticleContainer::Evolve (){
    WARPX_PROFILE("PPC::Evolve()");

    WARPX_PROFILE_VAR_NS("PPC::FieldGather", blp_fg);
    WARPX_PROFILE_VAR_START(blp_fg);
    FieldGather();
    WARPX_PROFILE_VAR_STOP(blp_fg);

}
```

Why bother with PRs?

Write a PR


Prepare a PR
- Communicate!
- One topic
- To the point
- Look at what's already there
Write good code
- To the point
- At the right place
- Modular
- Straightforward
- Comments
- No duplication
Test your code
- 2D local is not enough
- 2D/3D/RZ, CPU and GPU, serial and parallel, Threaded and not-threaded
Clean your PR
- Print statements
- Commented code
- Useless functions/variables
- Etc.
Comment your code
- To the point
- What's needed and not more
Write a PR description
- To the point
- What's needed and not more
- Labels
Ask for review
- Check the code
- Focus on what's tricky

WarpX: part II

GPU computing + parallelization
(how to use max_grid_size and blocking_factor)

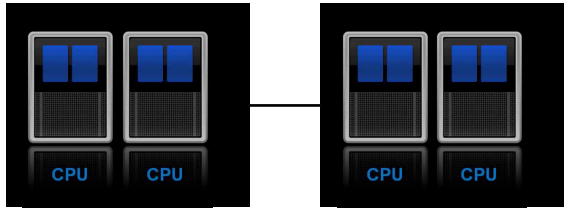
Maxence Thévenet
LBNL
03/12/2020

Part II: Performance and GPUs
- GPU computing
- Parallelization

GPU-accelerated supercomputing

CPU supercomputer
Cori @ NERSC (13/500)

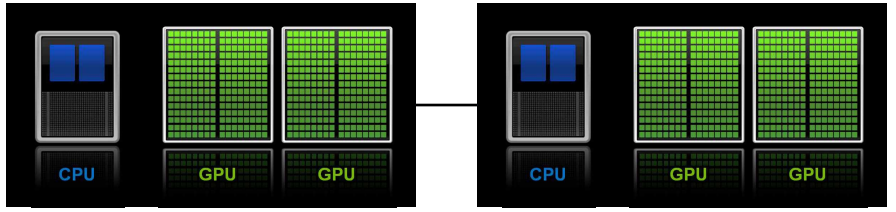
MPI + x



1 Cori node	1 Intel KNL CPU
3 TFLOPS 112 GB	3 TFLOPS 96 DDR4 + 16 GB HBM

GPU-accelerated supercomputer
Summit @ OLCF (1/500)

MPI + x

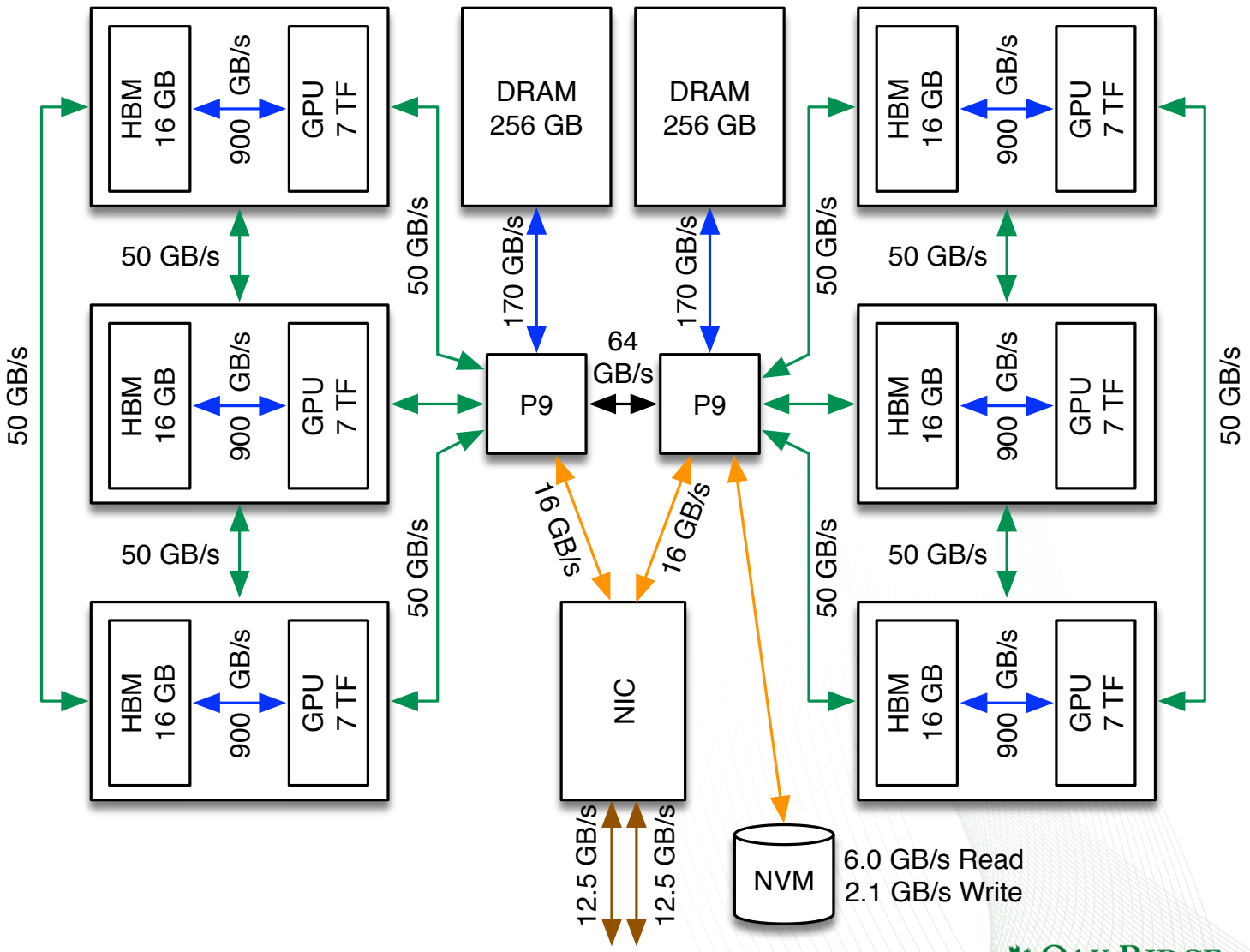


1 Summit node	2 IBM Power9 CPUs	6 NVIDIA V100 GPUs
42 TFLOPS 608 GB	0.8 TFLOPS 256 GB DDR4	7.8 TFLOPS 16 GB HBM

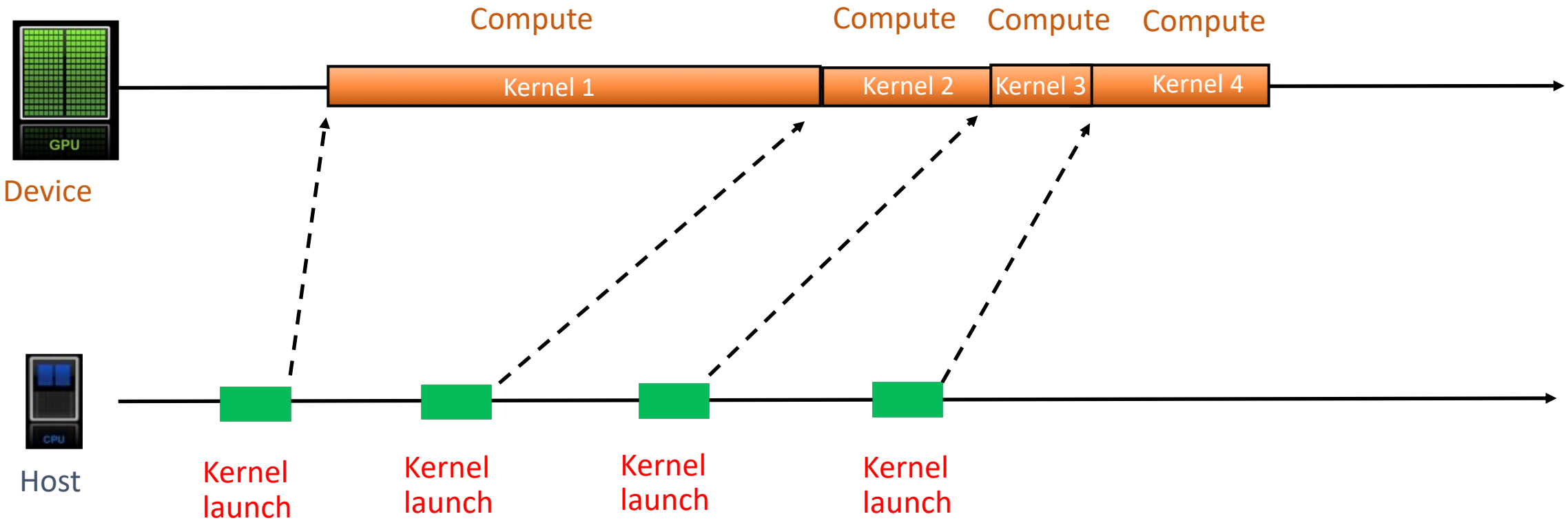
All pre-exascale and planned exascale supercomputers are accelerated

Summit Node Schematic

- Coherent memory across entire node
- NVLink v2 fully interconnects three GPUs and one CPU on each side of node
- PCIe Gen 4 connects NVM and NIC
- Single shared NIC with dual EDR ports



GPU-accelerated supercomputing

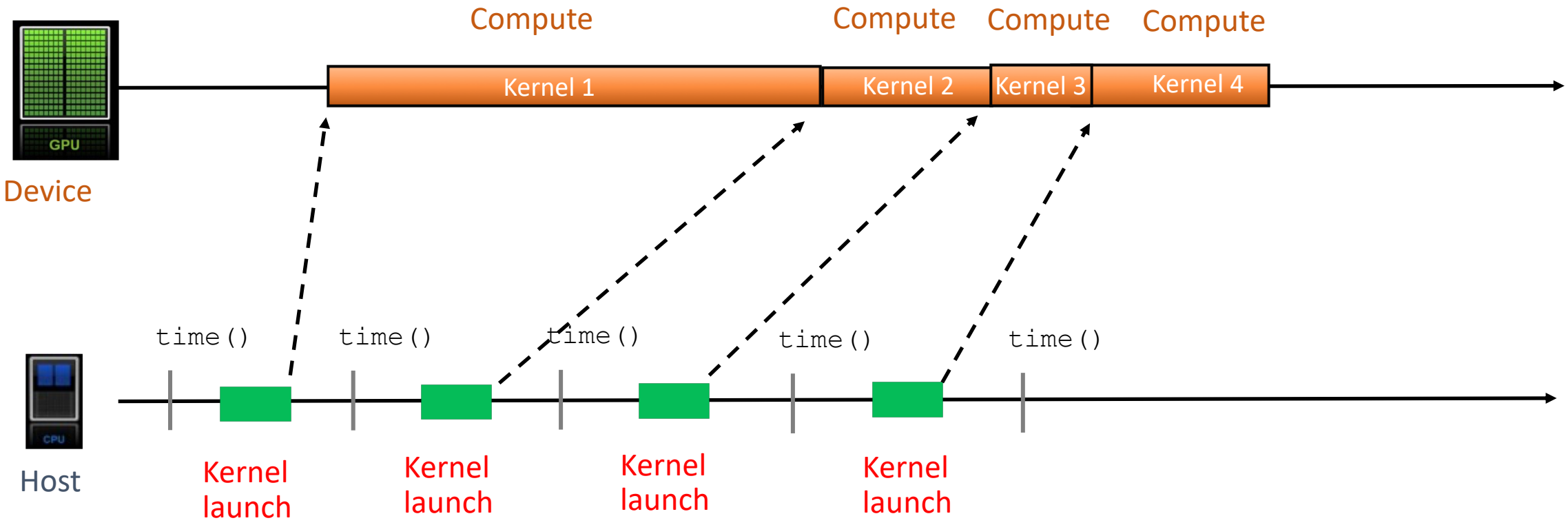


`ParallelFor()`

Asynchronous execution

You can explicitly synchronize with cudaDeviceSynchronize

GPU-accelerated supercomputing



Makes it difficult to have accurate timers

Now, where is the data?

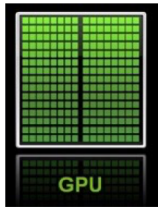
GPU-accelerated supercomputing

```
MPI_Init(&argc, &argv);

  amrex::Initialize(argc,argv);

    WarpX warpx;

    warpx.InitData();

    warpx.Evolve();

  amrex::Finalize();

MPI_Finalize();
```

- No explicit copying to GPU memory
- Pass raw pointers
- Keep data on GPU as much as possible
- Back to CPU if:
  - Out of memory
  - Communication/LB
  - Diagnostics
  - Any data access from the host…

Data initialized on the host, in unified memory.
Managed/Unified memory: single memory address space!

Host



Device

Host execution	Host memory 256 GB	Device execution	Device memory 16 GB
Allocate data	■		
Process data ← 170 GB/s →	■		
ParallelFor(data) = kernel launch	■	50 GB/s	
		I need data! pagefault	■
		Process data ← 900 GB/s →	■
Don't need data anymore.			■
			■

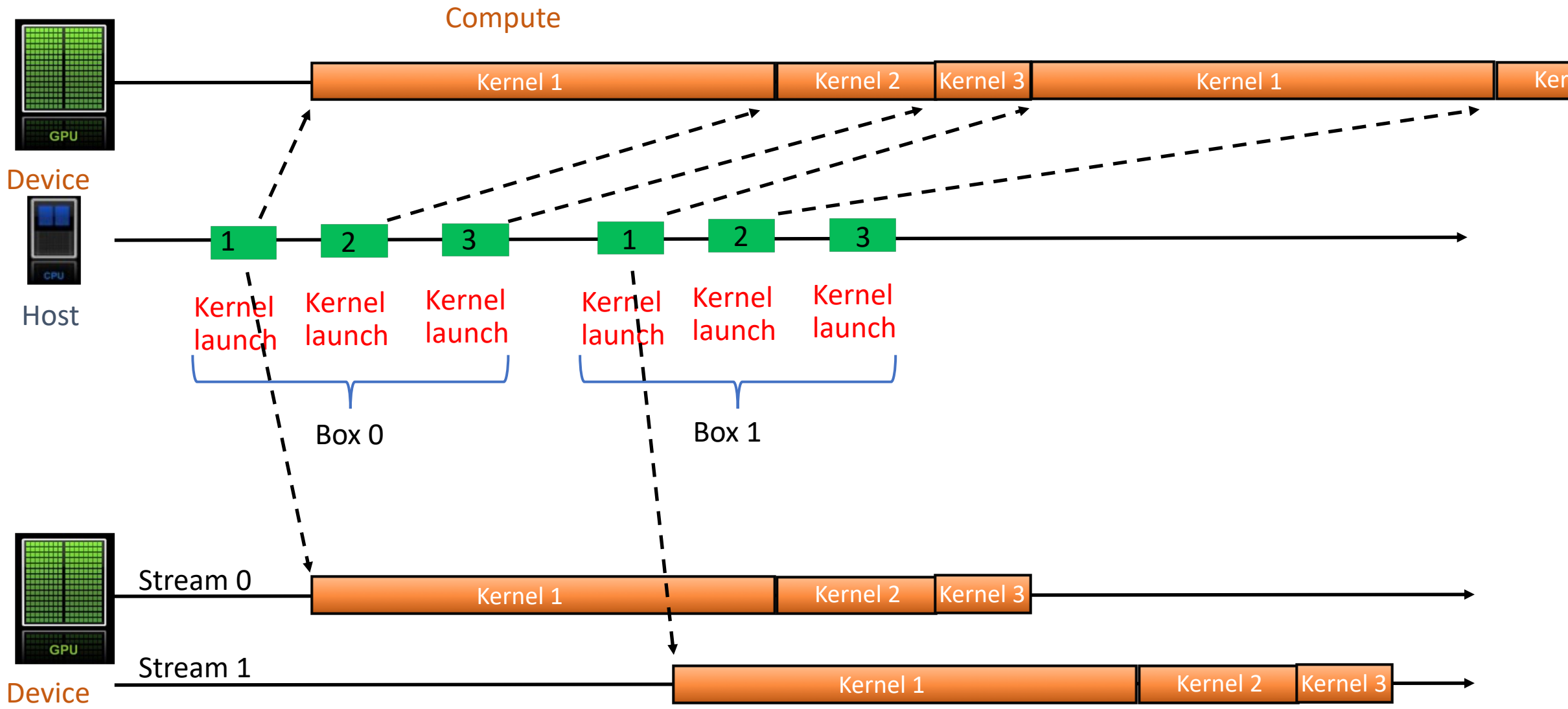
A detailed example

`warpx/FieldSolver/FiniteDifferenceSolver/EvolveB.cpp`

`amrex/Base/AMReX_Array4.H`

- What is Array4?
- Why #ifdef _OPENMP?
- Why TilingIfNotGPU?
- How many kernel launches?
- What exactly is copied to the GPU memory (if anything)?

MFIter, device synchronize and CUDA streams



CUDA streams allow for executing multiple kernels simultaneously on a GPU!

See AMReX GPU strategy https://amrex-codes.github.io/amrex/docs_html/GPU.html

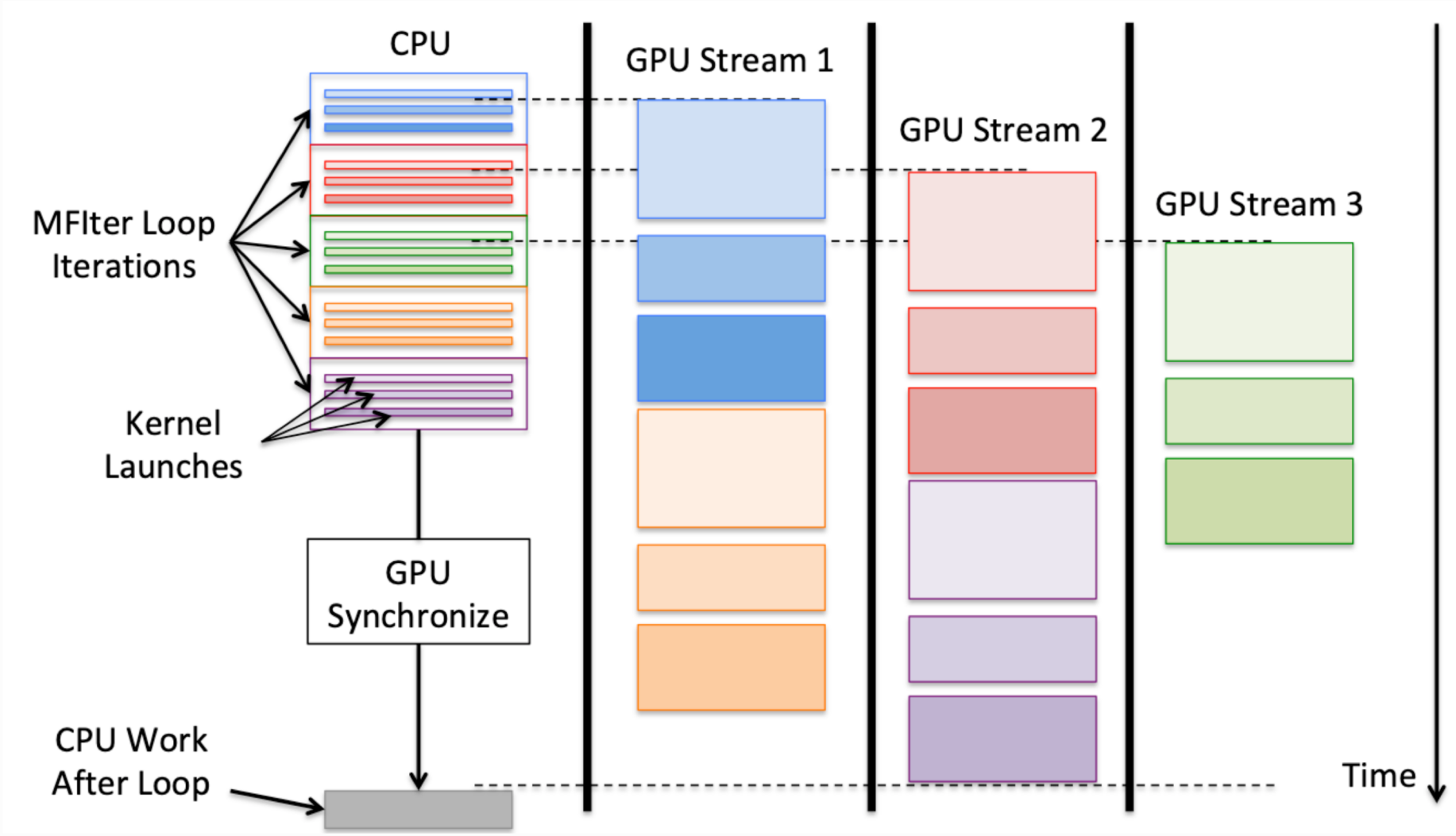


Fig. 16 Timeline illustration of GPU streams. Illustrates the case of an MFIter loop of five iterations with three GPU kernels each being ran with three GPU streams.

See AMReX GPU strategy https://amrex-codes.github.io/amrex/docs_html/GPU.html

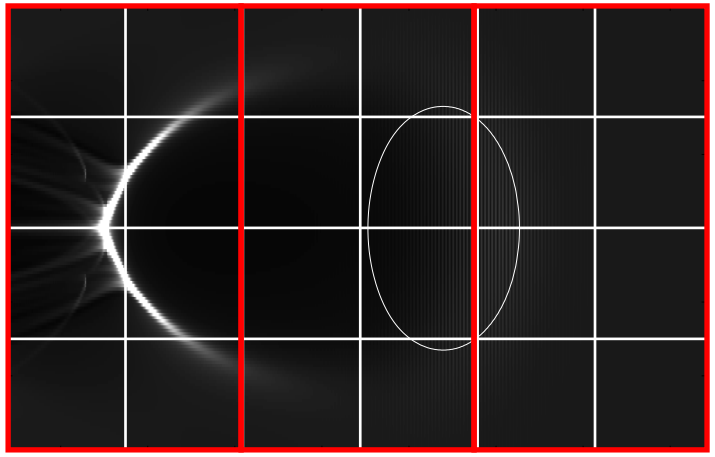
```
// See PhysicalParticleContainer::Evolve
for (WarpXParIter pti(*this, lev); pti.isValid(); ++pti){
    FArrayBox filtered_Ez;
    Elixir ezeli;
    const Box& tbox = …;
    filtered_Ez.resize(amrex::convert(tbox,Ez.box().ixType()));
    ezeli = filtered_Ez.Elixir();
    Filter[lev]->ApplyStencil(filtered_Ez, Ez[pti]);
    // Particles gather from filtered_Ez
}
```

Call ParIter (or MFIter) destructor
cudaDeviceSynchronize()

Parallelization

Grid = box

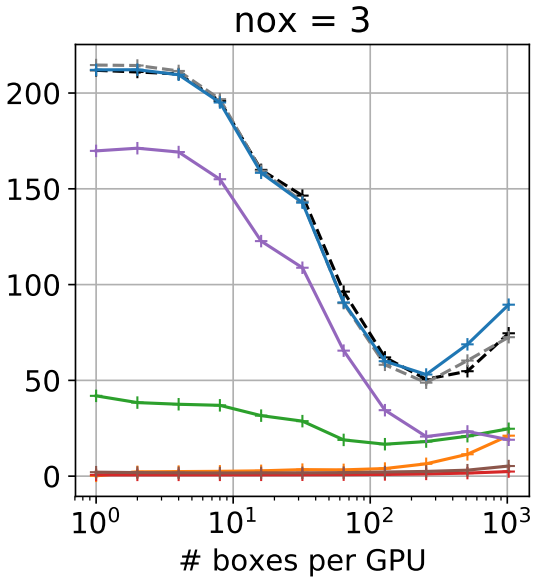
- `amr.max_grid_size` is the maximum number of points sdf per **grid** along each direction (default amr.max_grid_size=32 in 3D).
- `amr.blocking_factor`: The size of each **grid** must be divisible by the *blocking_factor* along all dimensions(default amr.blocking_factor=8).

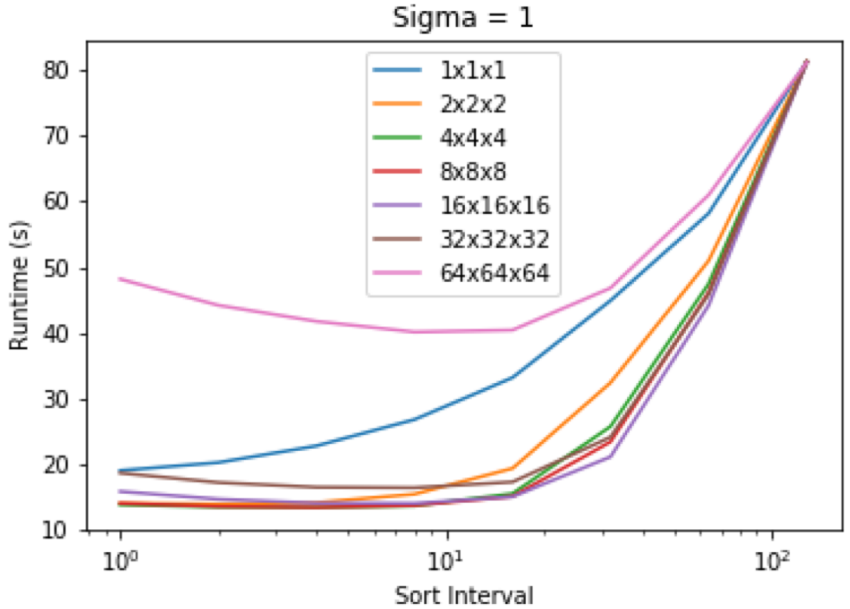


- the max_grid_size also has to be divisible by blocking_factor.
- The actual box decomposition depends on the number of MPI ranks requested.

For GPU:

- CUDA-aware MPI amrex.use_gpu_aware_mpi = 1
- We currently Synchronize at every WARPX_PROFILE call





Left: just using Tiny profiler data, we saw that sorting would accelerate the run.

For a lot of optimizations, the Tiny Profiler and the standard output give enough information:

- Problem well-decomposed?
- Problem load-balanced, LB?
- Problem fit in GPU memory?
- Problem/GPU big enough?
- Problem dominated by communications?

Run efficiently (on Summit, but also on any platform!)

- The standard output tells you the total number of GPUs used (so far we are using 1 MPI per GPU) and the number of boxes (called "grids") before the first iteration. Use it to make sure that you are using a few (1-10) boxes per GPU, this is how AMReX is meant to operate.
- A simulation executed efficiently should take < 1s / iteration on Summit, unless you are using the PSATD solver.
- If a run is very slow, the most likely cause is that **your simulation is running out of GPU memory** (see pagefaults), generating CPU-GPU transfers at each iteration. This is extremely inefficient, and WarpX is not meant to run this way. It usually means that you need to run on a larger number of nodes.
- If the simulations completes successfully, you can verify the memory utilization per GPUs with the lines (after Tiny Profiler timers)

  `Total GPU global memory (MB) spread across MPI: [16128 ... 16128]`

  → hardware limit (16 GB)

  `Free GPU global memory (MB) spread across MPI: [4 ... 6]`

  → GPU memory that is still available

  `[The Arena] space (MB) allocated spread across MPI: [24367 ... 27302]`

  → Memory allocated by the Arena (the AMReX module that handles memory allocation and management).

  As soon as Free GPU global memory is below 1000 (it will never be exactly 0), you may safely assume that you are overflowing the GPU memory. This is confirmed by the third line, which shows that the memory allocated is > 16 GB, so again this should run on more nodes.
- AMReX has an option to abort if the run exceeds GPU memory: **`amrex.abort_on_out_of_gpu_memory`**. I like it a lot.
- If a run has load imbalance, this can be seen in the Tiny Profiler output, for example in the current deposition time:

  `PPC::CurrentDeposition 1106 0 4.966 20.61 19.44%`

  meaning that 1 GPU is spending 0s in deposition, and another one is spending over 20s. In this case, note that load imbalance pollutes the timers for communication routines, so do not trust them too much. To fix this, you may use the AMReX load balancer with some of the options below (see the doc for all options, and ask Michael for more details). Feel free to try these on test2 (we didn't have time to cover it today)

```
# warpx.load_balance_int = 10
# algo. load_balance_costs_update = timers
# warpx.load_balance_with_sfc = 1
```