WarpX: a Particle-In-Cell code for the exascale era

Luca Fedeli

Paris, 07/02/2024
We are interested in several topics related to relativistic kinetic plasmas

- Strong-field QED in astrophysics (courtesy of R. Jambunathan)
- Strong-field QED in particle colliders (courtesy of A. Formenti)
- Strong-field QED in astrophysics (courtesy of R. Jambunathan)

- Laser-driven electron accelerators

Plasma mirror

Strong-field QED with Doppler-boosted laser-beams
WarpX: a Particle-In-Cell code for the exascale era

The Particle-In-Cell method

WarpX: a PIC code for the exascale era

AMREX: a framework for massively parallel, block-structured AMR applications
WarpX: a Particle-In-Cell code for the exascale era

The Particle-In-Cell method

WarpX: a PIC code for the exascale era

AMReX: a framework for massively parallel, block-structured AMR applications
A plasma is a “gas” of interacting charged particles.

The theoretical framework to model these plasmas is (collisionless) relativistic kinetic plasma theory.

Maxwell’s equations

\[ \nabla \cdot \mathbf{E} = \frac{\rho}{\varepsilon_0} \quad \nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t} \]
\[ \nabla \cdot \mathbf{B} = 0 \quad \nabla \times \mathbf{B} = \mu_0 \left( \mathbf{J} + \varepsilon_0 \frac{\partial \mathbf{E}}{\partial t} \right) \]

Vlasov’s equations

\[ \frac{\partial f_e}{\partial t} + \mathbf{v}_e \cdot \nabla f_e - \varepsilon_0 \left( \mathbf{E} + \mathbf{v}_e \times \mathbf{B} \right) \cdot \frac{\partial f_e}{\partial \mathbf{p}} = 0 \]
\[ \frac{\partial f_i}{\partial t} + \mathbf{v}_i \cdot \nabla f_i + Z_i \varepsilon_0 \left( \mathbf{E} + \mathbf{v}_i \times \mathbf{B} \right) \cdot \frac{\partial f_i}{\partial \mathbf{p}} = 0 \]
Particle-In-Cell codes are the tool of choice to model kinetic plasmas.
Particle-In-Cell codes are the tool of choice to model kinetic plasma phenomena.
Particle-In-Cell codes are the tool of choice to model kinetic plasma phenomena

Electromagnetic fields

Usually simulated with an FDTD solver

Field interpolation

Charged macro-particles

Current deposition
Particle-In-Cell codes are the tool of choice to model kinetic plasma phenomena.
Particle-In-Cell codes are the tool of choice to model kinetic plasma phenomena. Usually simulated with an FDTD solver, this is (typically) the most expensive part!
If we want to perform 3D simulations, we often end up needing a lot of computing power.
If we want to perform 3D simulations, we often end up needing a lot of computing power.

Physical problem

Solid foil (10 μm)

200 fs to simulate

Laser (λ = 800nm)

Numerical problem

50x50x50 μm³ box
10 nm resolution
10s particles per cell
~ 6000 steps
If we want to perform 3D simulations, we often end up needing a lot of computing power.

**Physical problem**
- Solid foil (10 μm)
- Laser (λ = 800nm)
- 200 fs to simulate

**Numerical problem**
- 50x50x50 μm³ box
- 10 nm resolution
- 10s particles per cell
- ~ 6000 steps

**Computational cost**
- Tens of Terabytes of RAM!
- Tens of thousands of CPU hours!
We can use domain decomposition to distribute a simulation over many computing nodes.
We can use domain decomposition to distribute a simulation over many computing nodes.
We can use domain decomposition to distribute a simulation over many computing nodes.
We need Particle-in-Cell codes able to run on the top supercomputers in the world

A 3D simulation may easily require tens of hours on several thousands of GPUs
WarpX: a Particle-In-Cell code for the exascale era

The Particle-In-Cell method

WarpX: a PIC code for the exascale era

AMReX: a framework for massively parallel, block-structured AMR applications
**WarpX** is a Particle-In-Cell code for the exascale era.

Open-source & multi-OS
Documentation: [ecp-warpx.github.io](https://ecp-warpx.github.io/)
WarpX is a Particle-In-Cell code for the exascale era.

Open-source & multi-OS
Documentation: ecp-warpx.github.io/

From your laptop to the largest supercomputers in the world!
WarpX is a Particle-In-Cell code for the exascale era.

Open-source & multi-OS
Documentation: ecp-warpx.github.io/

From your laptop to the largest supercomputers in the world!

Gordon Bell prize winner at Supercomputing 2022
WarpX is conceived & developed by a multidisciplinary, multi-institution team.
WarpX is now a project of the High-Performance Software foundation
WarpX is now a project of the High-Performance Software foundation
**WarpX** offers a very rich set of features

- Comprehensive additional physics modules set
- Cartesian & cylindrical geometries
- Advanced solvers
- Scalable output
- Advanced methods
- Ionization
- Collisions
- Nuclear fusion
- Strong-field QED
- 3D Cartesian grid
- Cylindrical grid (schematic)
- Z-order space filling curve
- Inside patch at \( L_{n=1} \): \( F_{n+1}(a) = (F_n(a) - F_n(0)) + F_{n+1}(0) \)
- Main grid: \( F_n(a) \)
WarpX offers a very rich set of features

Comprehensive additional physics modules set

Cartesian & cylindrical geometries

Scalable output

Advanced solvers

Advanced methods
Output of Particle-In-Cell simulations can be huge

For a large simulation (e.g., ~100% Fugaku or Frontier) a simulation snapshot may require:

- $1 \times 10^{12} - 1 \times 10^{13}$ particles $\rightarrow$ ~100-1000 TB for particles’ phase space
- $1 \times 10^{11} - 1 \times 10^{12}$ cells $\rightarrow$ ~1-10 TB for each field component
Main output strategy of WarpX relies on the Open Standard for Particle-Mesh Data (openPMD)

Open Standard for Particle-Mesh Data:
- high-level description
- minimal: users can add more
- human readable & machine actionable
- file format agnostic, portable
- scalable from desktop to supercomputer

openPMD standard (1.0.0, 1.0.1, 1.1.0)
the underlying file markup and definition
A Huebl et al., DOI:10.5281/zenodo.33624

openPMD-viewer
quick visualization
explore, e.g., in Jupyter

openPMD-api
reference library
file-format agnostic API
Main output strategy of WarpX relies on the Open Standard for Particle-Mesh Data (openPMD)

Open Standard for Particle-Mesh Data:
- high-level description
- minimal: users can add more
- human readable & machine actionable
- file format agnostic, portable
- scalable from desktop to supercomputer

openPMD standard (1.0.0, 1.0.1, 1.1.0)
the underlying file markup and definition
A Huebl et al., DOI:10.5281/zenodo.33624

openPMD-viewer
quick visualization
explore, e.g., in Jupyter

openPMD-api
reference library
file-format agnostic API

Supported back-ends:

openPMD-viewer
openPMD-api

Courtesy of Axel Huebl (LBNL)
Main output strategy of WarpX relies on the Open Standard for Particle-Mesh Data (openPMD)

Open Standard for Particle-Mesh Data:
- high-level description
- minimal: users can add more
- human readable & machine actionable
- file format agnostic, portable
- scalable from desktop to supercomputer

openPMD standard (1.0.0, 1.0.1, 1.1.0)
the underlying file markup and definition
A Huebl et al., DOI:10.5281/zenodo.33624

openPMD-viewer
quick visualization
explore, e.g., in Jupyter

openPMD-api
reference library
file-format agnostic API

Supported back-ends:

Courtesy of Axel Huebl (LBNL)
By using openPMD + ADIOS 2 we can achieve:

**Top performances:**
more than 5 Tbytes/s on 4096 Frontier nodes!

**On-the-fly data compression:**
ADIOS2 supports ZFP lossy compression and BZip2 lossless compression

A Huebl et al., “On the Scalability of Data Reduction Techniques in Current and Upcoming HPC Systems from an Application Perspective,” ISC High Performance Workshops, [DOI:10.1007/978-3-319-67630-2_2](https://doi.org/10.1007/978-3-319-67630-2_2) (2017)

**Data streaming to connect with other tools:**
e.g., in-situ visualization

WarpX allows the user to define one (or more) rectangular region where a Nx resolution is used.
WarpX allows the user to define one (or more) rectangular region where a Nx resolution is used.
WarpX allows the user to define one (or more) rectangular region where a \( N \times N \) resolution is used.
WarpX allows the user to define one (or more) rectangular region where a Nx resolution is used.
WarpX allows the user to define one (or more) rectangular region where a Nx resolution is used.
Mesh-refinement is very tricky in electromagnetic PIC codes

- Spurious self-forces
- Unphysical reflections

Higher res → higher max frequency!
Mesh-refinement is very tricky in electromagnetic PIC codes

Spurious self-forces

Unphysical reflections

Higher res → higher max frequency!

Mitigations

Buffer region (Solve field in buffer but gather force from underlying coarse parent grid)

PML to absorb spurious reflections

Inside patch at $L_{n+1}$:

$$F_{n+1}(a) = I[F_n(s) - F_{n+1}(c)] + F_{n+1}(f)$$

Main grid: $F_n(a)$
**WarpX** is used for many different applications!

- Plasma accelerators (LBNL, DESY, SLAC)
- Laser-ion acceleration - advanced mechanisms (LBNL)
- Plasma mirrors and high-field physics + QED (CEA Saclay/LBNL)
- Laser-ion acceleration - laser pulse shaping (LLNL)
- Fusion devices (Zap Energy, Avalanche Energy)
- Thermionic converter (Modern Electron)
- Pulsars, magnetic reconnection (LBNL)
- Magnetic fusion sheaths (LLNL)
- Microelectronics (LBNL) - ARTEMIS

[Image of various applications with examples and diagrams]
WarpX is used for tests & production runs on a variety of HPC systems.
Let's have a look at WarpX performances
WarpX performances:
Floating Point Operations per Second
In a Particle-In-Cell code, the main kernels typically have a low arithmetic intensity → performances limited by memory

Roofline analysis carried out in 2021 on NVIDIA V100 GPU →

We expect to achieve only few % of the peak FLOP/s efficiency

<table>
<thead>
<tr>
<th>System</th>
<th>DP PFlop/s</th>
<th>Linpack Benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perlmutter A100</td>
<td>3.38</td>
<td>12.9%</td>
</tr>
<tr>
<td>Summit V100</td>
<td>11.79</td>
<td>8.3%</td>
</tr>
</tbody>
</table>
We expect to achieve **only few % of the peak FLOP/s efficiency**

<table>
<thead>
<tr>
<th>System</th>
<th>DP PFlop/s</th>
<th>Linpack Benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perlmutter</td>
<td>3.38</td>
<td>12.9%</td>
</tr>
<tr>
<td>NVIDIA A100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Summit</td>
<td>11.79</td>
<td>8.3%</td>
</tr>
<tr>
<td>NVIDIA V100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frontier</td>
<td>43.45</td>
<td>3.3%</td>
</tr>
<tr>
<td>AMD MI250X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Nov 2022
We expect to achieve **only few % of** the peak FLOP/s efficiency

<table>
<thead>
<tr>
<th>System</th>
<th>DP PFlop/s</th>
<th>Linpack Benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perlmutter</td>
<td>3.38</td>
<td>12.9%</td>
</tr>
<tr>
<td>NVIDIA A100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Summit</td>
<td>11.79</td>
<td>8.3%</td>
</tr>
<tr>
<td>NVIDIA V100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frontier</td>
<td>43.45</td>
<td>3.3%</td>
</tr>
<tr>
<td>AMD MI250X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fugaku</td>
<td>5.31</td>
<td>1.1%</td>
</tr>
<tr>
<td>Fujitsu A64FX</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Specific tuning</strong></td>
<td><strong>SP: 17.3</strong></td>
<td><strong>x3.3</strong></td>
</tr>
</tbody>
</table>

Nov 2022
WarpX performances:

Weak Scaling
Very good weak scaling over 4-5 orders of magnitude

**Nodes**
Frontier: 1 - 8,576 (pre-acceptance)
Fugaku: 1 - 152,064
Summit: 2 - 4,263
Perlmutter: 1 - 1,088 (pre-acceptance)

L.Fedeli et al. SC22 proceedings (2022)
WarpX performances:

**Strong Scaling**
WarpX can be strong-scaled by one order of magnitude

Nodes
- Frontier: 512 – 8,192 (pre-acceptance)
- Fugaku: 6,144 - 152,064
- Summit: 512 - 4,096
- Perlmutter: 15 – 480 (pre-acceptance)

L.Fedeli et al. SC22 proceedings (2022)
WarpX: a Particle-In-Cell code for the exascale era

The Particle-In-Cell method

WarpX: a PIC code for the exascale era

AMREX: a framework for massively parallel, block-structured AMR applications
WarpX is built on top of the open-source AMReX library

AMReX
Containers, Communication, Portability, Utilities

Python: Modules, PICMI interface, Workflows

WarpX
full PIC, LPA/LPI

HiPACE++
quasi-static, PWFA

ARTEMIS
microelectronics

ImpactX
accelerator lattice design

Object-Level Python Bindings extensible, AI/ML

pyAMReX
PICSAR
QED Modules

ABLASTR library: common PIC physics

Diagnostics
I/O code coupling

openPMD
ADIO S2
AAD
ZFP

FFT
on- or multi-device

Lin. Alg.
BLAS++ LAPACK++

MPI

CUDA, OpenMP, SYCL, HIP
AMReX – A Software Framework for Adaptive Mesh Refinement

Basic features of AMReX
- Parallel mesh and particle data structures and iterators
- Inter-node communication, dynamic load balancing of heterogeneous workloads
- Efficient embedded boundary discretizations for complex geometry
- Linear solvers for cell-centered and nodal data with optional EB discretizations and multiple AMR levels

Hybrid model for parallelism
- MPI for coarse-grained distribution of field and particle data on grid patches to nodes
- CUDA / HIP / SYCL for fine-grained parallelism on GPUs. OpenMP for multicore CPUs
- Light-weight abstraction layers hides specific architecture from applications – application runs on different architectures without code modification

Courtesy of Weiqun Zhang (LBNL)
AMReX supported seven ECP and non-ECP applications

**Astrophysics:**
- Castro (compressible)
- MaestroEx (low-Mach)
- SedonaEx (Monte Carlo radiation transport)
- Emu (neutrino transport)
- Quokka (radiation-hydrodynamics)

**Cosmology:**
- Nyx

**Combustion:**
- PeleC (Compressible)
- PeleLM (Low Mach)

**Accelerator Modelling:**
- WarpX
- ImpactX
- Hipace++

**Magnetically-confined fusion:**
- GEMPIC

**Multi-scale Modelling and Stochastic Systems:**
- FHDeX

**Electromagnetics:**
- ARTEMIS

**Epidemiology:**
- Exa-Epi

**Incompressible / Compressible Navier-Stokes:**
- IAMR / CAMR
- incflo

**Solid Mechanics:**
- Alamo

**Biological cell modelling:**
- BoltzmanMFX
- CCM

**Multi-phase Flow:**
- MFIX-Exa

**Atmospheric science:**
- AMR-wind
- ERF

**Ocean modeling:**
- ROMS-X

Courtesy of Weiqun Zhang (LBNL)
AMReX is also a project of the High-Performance Software foundation.
AMReX provides **Mesh Data Structures** and **Particle Data Structures**.
AMReX provides Tools for Parallel communications and for Load Balancing
AMReX provides a performance portability layer

A “Cambrian explosion” of computing architectures is a challenge for HPC software!
AMReX provides a performance portability layer

Write the code **once**, specialize at **compile-time**

ParallelFor(/Scan/Reduce)

```cpp
amrex::ParallelFor( n_particles,
    [=] AMREX_GPU_DEVICE (long i) {
        UpdatePosition( x[i], y[i], z[i],
            ux[i], uy[i], uz[i], dt );
    });
```
Let’s consider a toy Heat Equation solver written using AMReX

Heat equation

\[ \frac{\partial \phi}{\partial t} = \nabla^2 \phi \]
Let’s consider a toy Heat Equation solver written using AMReX

Heat equation

$$\frac{\partial \phi}{\partial t} = \nabla^2 \phi$$
Let’s consider a toy Heat Equation solver written using AMReX

Heat equation

\[ \frac{\partial \phi}{\partial t} = \nabla^2 \phi \]
Let’s consider a toy Heat Equation solver written using AMReX

Heat equation

\[ \frac{\partial \phi}{\partial t} = \nabla^2 \phi \]
Let’s consider a toy Heat Equation solver written using AMReX

**Heat equation**

\[ \frac{\partial \phi}{\partial t} = \nabla^2 \phi \]
AMReX subdivides the simulation domain into “boxes”, and boxes can be further subdivided into “tiles”.

Without tiling

With tiling

Each MPI task manages one or more boxes. Tiles are usually used only on CPUs for data locality & OpenMP parallelization.
AMReX provides a parallel data-structure (MultiFab) to manipulate multi-dimensional arrays.

\[
\frac{\partial \phi}{\partial t} = \nabla^2 \phi
\]

```cpp
amrex::MultiFab phi_old(
    box_array,
    distribution_mapping,
    Ncomp, // Ncomp=1
    Nghost); // Nghost=1

amrex::MultiFab phi_new(
    box_array,
    distribution_mapping,
    Ncomp, // Ncomp=1
    Nghost); // Nghost=1
```

Adapted from AMReX tutorials
AMReX provides a parallel data-structure (MultiFab) to manipulate multi-dimensional arrays

\[ \frac{\partial \phi}{\partial t} = \nabla^2 \phi \]

A one-component field

\[ \phi \]

Adapted from AMReX tutorials
AMReX provides a parallel data-structure (MultiFab) to manipulate multi-dimensional arrays

\[
\frac{\partial \phi}{\partial t} = \nabla^2 \phi
\]

A one-component field, over a collection of boxes

\[
\phi
\]

Adapted from AMReX tutorials

amrex::MultiFab phi_old(
    box_array,
    distribution_mapping,
    Ncomp, // Ncomp=1
    Nghost); // Nghost=1

amrex::MultiFab phi_new(
    box_array,
    distribution_mapping,
    Ncomp, // Ncomp=1
    Nghost); // Nghost=1
AMReX provides a parallel data-structure (MultiFab) to manipulate multi-dimensional arrays

```
amrex::MultiFab phi_old(
    box_array,
    distribution_mapping,
    Ncomp, // Ncomp=1
    Nghost); // Nghost=1

amrex::MultiFab phi_new(
    box_array,
    distribution_mapping,
    Ncomp, // Ncomp=1
    Nghost); // Nghost=1
```

A one-component field, over a collection of boxes, distributed across N MPI tasks

\[
\frac{\partial \phi}{\partial t} = \nabla^2 \phi
\]
AMReX provides a parallel data-structure (MultiFab) to manipulate multi-dimensional arrays.

\[
\frac{\partial \phi}{\partial t} = \nabla^2 \phi
\]

A one-component field, over a collection of boxes, distributed across N MPI tasks, using 1 ghost cell.

```
amrex::MultiFab phi_old(
    box_array,
    distribution_mapping,
    Ncomp, // Ncomp=1
    Nghost); // Nghost=1
```

```
amrex::MultiFab phi_new(
    box_array,
    distribution_mapping,
    Ncomp, // Ncomp=1
    Nghost); // Nghost=1
```

Adapted from AMReX tutorials
AMReX provides a parallel data-structure (MultiFab) to manipulate multi-dimensional arrays

\[
\frac{\partial \phi}{\partial t} = \nabla^2 \phi
\]

A one-component field, over a collection of boxes, distributed across N MPI tasks, using 1 ghost cell

```
amrex::MultiFab phi_old(
    box_array,
    distribution_mapping,
    Ncomp, // Ncomp=1
    Nghost); // Nghost=1
```

```
amrex::MultiFab phi_new(
    box_array,
    distribution_mapping,
    Ncomp, // Ncomp=1
    Nghost); // Nghost=1
```

Adapted from AMReX tutorials
Let's have a look at the main kernel of the toy Heat Equation solver written using AMReX

Adapted from AMReX tutorials

\[ \frac{\partial \phi}{\partial t} = \nabla^2 \phi \]

```c
for (int step = 1; step <= nsteps; ++step) {
    phi_new.FillBoundary(amrex::IntVect{Nghost,Nghost,Nghost}, geom.periodicity());
    amrex::MultiFab::Swap(phi_old, phi_new, 0, 0, Ncomp, Nghost);

    #ifdef AMREX_USE_OMP
    #pragma omp parallel
    #endif
    for (amrex::MFIter mfi(phi_old, amrex::TilingIfNotGPU()); mfi.isValid(); ++mfi ){
        const amrex::Box& bx = mfi.tilebox();
        const amrex::Array4<amrex::Real const>& phi_old_array = phi_old.const_array(mfi);
        const amrex::Array4<amrex::Real      >& phi_new_array = phi_new.array(mfi);

        amrex::ParallelFor(bx, [=] AMREX_GPU_DEVICE (int i, int j, int k){
            phi_new_array(i,j,k) = phi_old_array(i,j,k) +
            (phi_old_array(i+1,j,k) - 2.*phi_old_array(i,j,k) + phi_old_array(i-1,j,k)) * dt_over_dx0dx0
            +(phi_old_array(i,j+1,k) - 2.*phi_old_array(i,j,k) + phi_old_array(i,j-1,k)) * dt_over_dx1dx1
            +(phi_old_array(i,j,k+1) - 2.*phi_old_array(i,j,k) + phi_old_array(i,j,k-1)) * dt_over_dx2dx2);
        });
    }
    time += dt;
}
```
Let’s have a look at the main kernel of the toy Heat Equation solver written using AMReX

\[ \frac{\partial \phi}{\partial t} = \nabla^2 \phi \]

Adapted from AMReX tutorials

```c
for (int step = 1; step <= nsteps; ++step)
{
    phi_new.FillBoundary(amrex::IntVect{Nghost,Nghost,Nghost}, geom.periodicity());
    amrex::MultiFab::Swap(phi_old, phi_new, 0, 0, Ncomp, Nghost);

#ifdef AMREX_USE_OMP
    #pragma omp parallel
#endif
    for (amrex::MFIter mfi(phi_old, amrex::TilingIfNotGPU()); mfi.isValid(); ++mfi )
    {
        const amrex::Box& bx = mfi.tilebox();
        const amrex::Array4<amrex::Real const>& phi_old_array = phi_old.const_array(mfi);
        const amrex::Array4<amrex::Real      >& phi_new_array = phi_new.array(mfi);

        amrex::ParallelFor(bx, [=] AMREX_GPU_DEVICE (int i, int j, int k){
            phi_new_array(i,j,k) = phi_old_array(i,j,k) +
            ( (phi_old_array(i+1,j,k) - 2.*phi_old_array(i,j,k) + phi_old_array(i-1,j,k)) * dt_over_dx0dx0
             + (phi_old_array(i,j+1,k) - 2.*phi_old_array(i,j,k) + phi_old_array(i,j-1,k)) * dt_over_dx1dx1
             + (phi_old_array(i,j,k+1) - 2.*phi_old_array(i,j,k) + phi_old_array(i,j,k-1)) * dt_over_dx2dx2);});
    }
    time += dt;
}
```

This is just the main loop over the timesteps of the simulation.
Let’s have a look at the main kernel of the toy Heat Equation solver written using AMReX

Adapted from AMReX tutorials

\[
\frac{\partial \phi}{\partial t} = \nabla^2 \phi
\]

```c
for (int step = 1; step <= nsteps; ++step){
    phi_new.FillBoundary(amrex::IntVect{Nghost,Nghost,Nghost}, geom.periodicity());
    amrex::MultiFab::Swap(phi_old, phi_new, 0, 0, Ncomp, Nghost);

    #ifdef AMREX_USE_OMP
    #   pragma omp parallel
    #endif
    for (amrex::MFIter mfi(phi_old, amrex::TilingIfNotGPU()); mfi.isValid(); ++mfi ){
        const amrex::Box& bx = mfi.tilebox();
        const amrex::Array4<amrex::Real const>& phi_old_array = phi_old.const_array(mfi);
        const amrex::Array4<amrex::Real      >& phi_new_array = phi_new.array(mfi);

        amrex::ParallelFor(bx, [=] AMREX_GPU_DEVICE (int i, int j, int k){
            phi_new_array(i,j,k) = phi_old_array(i,j,k) +
                ( (phi_old_array(i+1,j,k) - 2.*phi_old_array(i,j,k) + phi_old_array(i-1,j,k)) * dt_over_dx0dx0
                +(phi_old_array(i,j+1,k) - 2.*phi_old_array(i,j,k) + phi_old_array(i,j-1,k)) * dt_over_dx1dx1
                +(phi_old_array(i,j,k+1) - 2.*phi_old_array(i,j,k) + phi_old_array(i,j,k-1)) * dt_over_dx2dx2);});
    }
    time += dt;
}
```

AMReX manages the filling of the ghost cells of the \( \varphi \) data structure.
Let's have a look at the main kernel of the toy Heat Equation solver written using AMReX

\[ \frac{\partial \phi}{\partial t} = \nabla^2 \phi \]

Adapted from AMReX tutorials

```cpp
for (int step = 1; step <= nsteps; ++step)
{
    phi_new.FillBoundary(amrex::IntVect{Nghost,Nghost,Nghost}, geom.periodicity());
    amrex::MultiFab::Swap(phi_old, phi_new, 0, 0, Ncomp, Nghost);

    #ifdef AMREX_USE_OMP
    #   pragma omp parallel
    #endif
    for (amrex::MFIter mfi(phi_old, amrex::TilingIfNotGPU()); mfi.isValid(); ++mfi ){
        const amrex::Box& bx = mfi.tilebox();
        const amrex::Array4<amrex::Real const>& phi_old_array = phi_old.const_array(mfi);
        const amrex::Array4<amrex::Real      >& phi_new_array = phi_new.array(mfi);

        amrex::ParallelFor(bx, [=] AMREX_GPU_DEVICE (int i, int j, int k){
            phi_new_array(i,j,k) = phi_old_array(i,j,k) +
            ( (phi_old_array(i+1,j,k) - 2.*phi_old_array(i,j,k) + phi_old_array(i-1,j,k)) * dt_over_dx0dx0
             +(phi_old_array(i,j+1,k) - 2.*phi_old_array(i,j,k) + phi_old_array(i,j-1,k)) * dt_over_dx1dx1
             +(phi_old_array(i,j,k+1) - 2.*phi_old_array(i,j,k) + phi_old_array(i,j,k-1)) * dt_over_dx2dx2));
        });
    time += dt;
}
```
Let's have a look at the main kernel of the toy Heat Equation solver written using AMReX

Adapted from AMReX tutorials

\[
\frac{\partial \phi}{\partial t} = \nabla^2 \phi
\]

```cpp
for (int step = 1; step <= nsteps; ++step)
{
    phi_new.FillBoundary(amrex::IntVect{Nghost,Nghost,Nghost}, geom.periodicity());
    amrex::MultiFab::Swap(phi_old, phi_new, 0, 0, Ncomp, Nghost);

    #ifdef AMREX_USE_OMP
    #   pragma omp parallel
    #endif
    for (amrex::MFIter mfi(phi_old, amrex::TilingIfNotGPU()); mfi.isValid(); ++mfi ){
        const amrex::Box& bx = mfi.tilebox();
        const amrex::Array4<amrex::Real const>& phi_old_array = phi_old.const_array(mfi);
        const amrex::Array4<amrex::Real      >& phi_new_array = phi_new.array(mfi);
        amrex::ParallelFor(bx, [=] AMREX_GPU_DEVICE (int i, int j, int k){
            phi_new_array(i,j,k) = phi_old_array(i,j,k) +
            ( (phi_old_array(i+1,j,k) - 2.*phi_old_array(i,j,k) + phi_old_array(i-1,j,k)) * dt_over_dx0dx0
            + (phi_old_array(i,j+1,k) - 2.*phi_old_array(i,j,k) + phi_old_array(i,j-1,k)) * dt_over_dx1dx1
            + (phi_old_array(i,j,k+1) - 2.*phi_old_array(i,j,k) + phi_old_array(i,j,k-1)) * dt_over_dx2dx2);});
    }
    time += dt;
}
```

These lines acquire the right region of the data structures containing old and new \( \varphi \) data.
Let’s have a look at the main kernel of the toy Heat Equation solver written using AMReX

Adapted from AMReX tutorials

\[
\frac{\partial \phi}{\partial t} = \nabla^2 \phi
\]

for (int step = 1; step <= nsteps; ++step){
    phi_new.FillBoundary(amrex::IntVect{Nghost,Nghost,Nghost}, geom.periodicity());
    amrex::MultiFab::Swap(phi_old, phi_new, 0, 0, Ncomp, Nghost);
    
    #ifdef AMREX_USE_OMP
    #   pragma omp parallel
    #endif
    for (amrex::MFIter mfi(phi_old, amrex::TilingIfNotGPU()); mfi.isValid(); ++mfi ){
        const amrex::Box& bx = mfi.tilebox();
        const amrex::Array4<amrex::Real const>& phi_old_array = phi_old.const_array(mfi);
        const amrex::Array4<amrex::Real      >& phi_new_array = phi_new.array(mfi);

        #amrex::ParallelFor(bx, [=] AMREX_GPU_DEVICE (int i, int j, int k){
            phi_new_array(i,j,k) = phi_old_array(i,j,k) +
                ( phi_old_array(i+1,j,k) - 2.*phi_old_array(i,j,k) + phi_old_array(i-1,j,k) ) * dt_over_dx0dx0
            + (phi_old_array(i,j+1,k) - 2.*phi_old_array(i,j,k) + phi_old_array(i,j-1,k) ) * dt_over_dx1dx1
            + (phi_old_array(i,j,k+1) - 2.*phi_old_array(i,j,k) + phi_old_array(i,j,k-1) ) * dt_over_dx2dx2);
    } time += dt;
}
Let’s have a look at the main kernel of the toy Heat Equation solver written using AMReX

Adapted from AMReX tutorials

\[ \frac{\partial \phi}{\partial t} = \nabla^2 \phi \]

```c
for (int step = 1; step <= nsteps; ++step){
    phi_new.FillBoundary(amrex::IntVect{Nghost,Nghost,Nghost}, geom.periodicity());
    amrex::MultiFab::Swap(phi_old, phi_new, 0, 0, Ncomp, Nghost);
    #ifdef AMREX_USE_OMP
    #   pragma omp parallel
    #endif
    for (amrex::MFIter mfi(phi_old, amrex::TilingIfNotGPU()); mfi.isValid(); ++mfi ){
        const amrex::Box& bx = mfi.tilebox();
        const amrex::Array4<amrex::Real const>& phi_old_array = phi_old.const_array(mfi);
        const amrex::Array4<amrex::Real      >& phi_new_array = phi_new.array(mfi);
        amrex::ParallelFor(bx, [=] AMREX_GPU_DEVICE (int i, int j, int k){
            phi_new_array(i,j,k) = phi_old_array(i,j,k) +
            ( (phi_old_array(i+1,j,k) - 2.*phi_old_array(i,j,k) + phi_old_array(i-1,j,k)) * dt_over_dx0dx0
             +(phi_old_array(i,j+1,k) - 2.*phi_old_array(i,j,k) + phi_old_array(i,j-1,k)) * dt_over_dx1dx1
             +(phi_old_array(i,j,k+1) - 2.*phi_old_array(i,j,k) + phi_old_array(i,j,k-1)) * dt_over_dx2dx2);});
    }
    time += dt;
}
```

This is needed to be able to compile the kernel for GPUs.
Let’s have a look at the main kernel of the toy Heat Equation solver written using AMReX

```
for (int step = 1; step <= nsteps; ++step){
    phi_new.FillBoundary(amrex::IntVect{Nghost,Nghost,Nghost}, geom.periodicity());
    amrex::MultiFab::Swap(phi_old, phi_new, 0, 0, Ncomp, Nghost);

    #ifdef AMREX_USE_OMP
    #   pragma omp parallel
    #endif
    for (amrex::MFIter mfi(phi_old, amrex::TilingIfNotGPU()); mfi.isValid(); ++mfi ){
        const amrex::Box& bx = mfi.tilebox();
        const amrex::Array4<amrex::Real const>& phi_old_array = phi_old.const_array(mfi);
        const amrex::Array4<amrex::Real      >& phi_new_array = phi_new.array(mfi);
        amrex::ParallelFor(bx, [=] AMREX_GPU_DEVICE (int i, int j, int k){
            phi_new_array(i,j,k) = phi_old_array(i,j,k) +
                ( (phi_old_array(i+1,j,k) - 2.*phi_old_array(i,j,k) + phi_old_array(i-1,j,k)) * dt_over_dx0dx0
                +(phi_old_array(i,j+1,k) - 2.*phi_old_array(i,j,k) + phi_old_array(i,j-1,k)) * dt_over_dx1dx1
                +(phi_old_array(i,j,k+1) - 2.*phi_old_array(i,j,k) + phi_old_array(i,j,k-1)) * dt_over_dx2dx2);});
    }
    time += dt;
}
```

This actually computes:

\[
\frac{\partial \phi}{\partial t} = \nabla^2 \phi
\]
And at compile time (with cmake or GNUmake):

NVIDIA GPUs → \( \text{USE\_CUDA} = \text{TRUE} \)

AMD GPUs → \( \text{USE\_HIP} = \text{TRUE} \)

CPUs → \( \text{USE\_OMP} = \text{TRUE} \)

Intel GPUs → \( \text{USE\_SYCL} = \text{TRUE} \)
Conclusions:

The AMReX library provides a performance portability layer and building blocks to manage fields and particles.

WarpX is a state-of-the-art Particle-In-Cell code conceived for the exascale era and suitable for a variety of applications.

https://amrex-codes.github.io/amrex/

https://ecp-warpx.github.io/warpx/